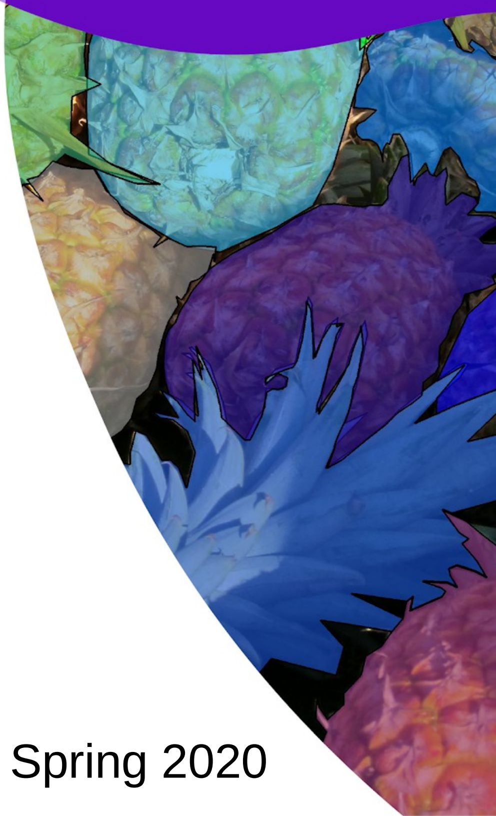
NEW YORK UNIVERSITY

Backpropagation

http://bit.ly/DLSP20



Yann LeCun
NYU - Courant Institute & Center for Data Science
Facebook AI Research
http://yann.lecun.com
TAs: Alfredo Canziani, Mark Goldstein

Deep Learning, NYU, Spring 2020

Parameterized Model

- **Parameterized model**

$$\bar{y} = G(x, w)$$

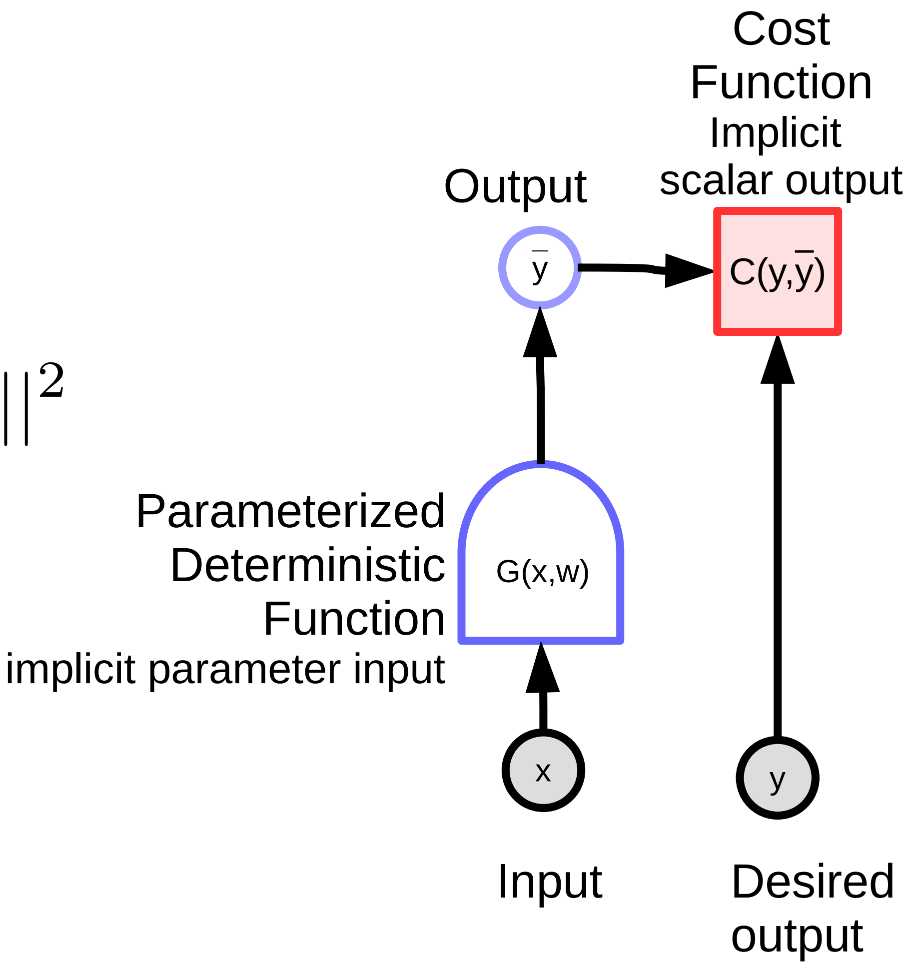
- Example: linear regression

$$\bar{y} = \sum_i w_i x_i \quad C(y, \bar{y}) = ||y - \bar{y}||^2$$

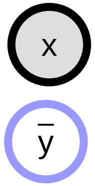
- Example: Nearest neighbor:

$$\bar{y} = \text{argmin}_k ||x - w_{k,.}||^2$$

- Computing function G may involve complicated algorithms

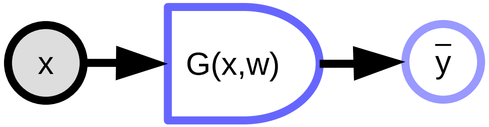


Block diagram notations for computation graphs

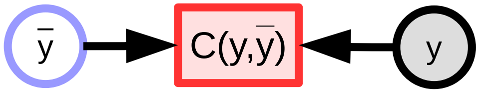


- **Variables (tensor, scalar, continuous, discrete...)**
  - Observed: input, desired output…
  - Computed variable: outputs of deterministic functions

easier to compute from x to y.



- **Deterministic function**
  - Multiple inputs and outputs (tensors, scalars,….)
  - Implicit parameter variable (here: w)



- **Scalar-valued function (implicit output)**
  - Single scalar output (implicit)
  - used mostly for cost functions

Loss function, average loss.

- **Simple per-sample loss function**

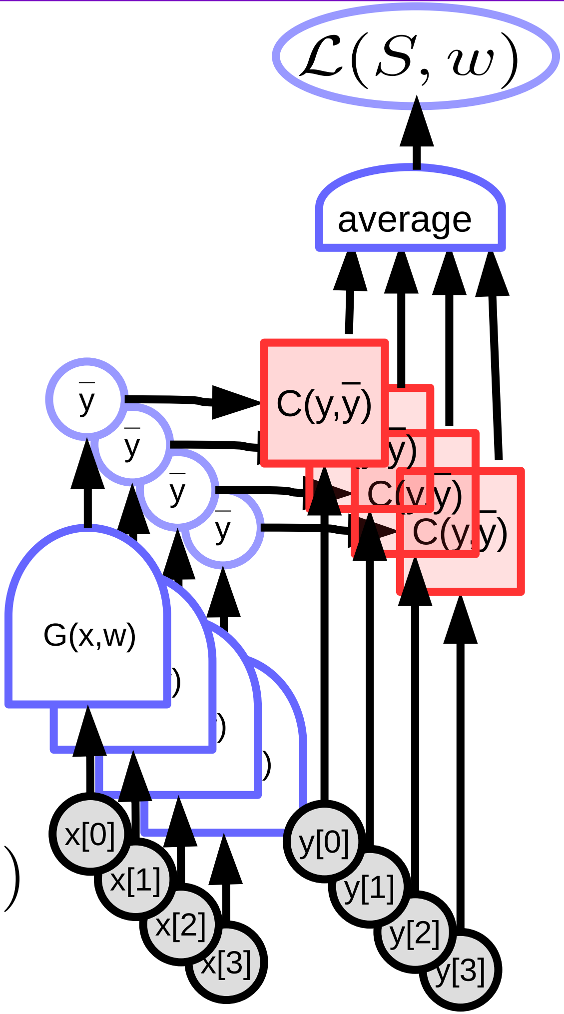
$$L(x, y, w) = C(y, G(x, w))$$

- **A set of samples**

$$S = \{(x[p], y[p]) \ / \ p = 0 \ldots P-1\}$$

- **Average loss over the set**

$$\mathcal{L}(S, w) = \frac{1}{P} \sum_{(x,y)} L(x, y, w) = \frac{1}{P} \sum_{p=0}^{P-1} L(x[p], y[p], w)$$



Gradient Descent

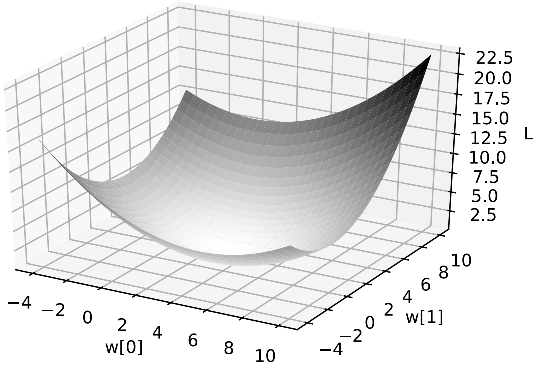
- **Full (batch) gradient**

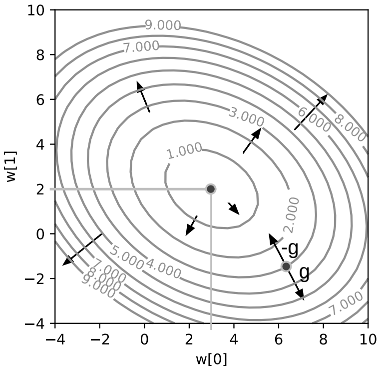
$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(S, w)}{\partial w}$$

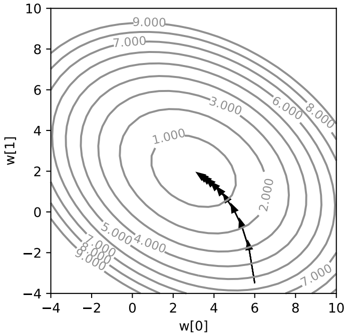
- **Stochastic Gradient (SGD)**
  - Pick a p in 0...P-1, then update w:

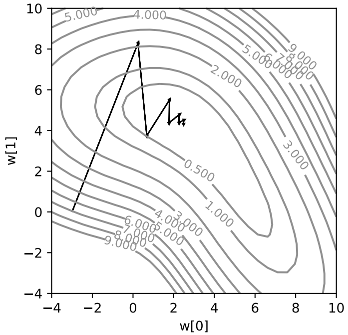
$$w \leftarrow w - \eta \frac{\partial L(x[p], y[p], w)}{\partial w}$$

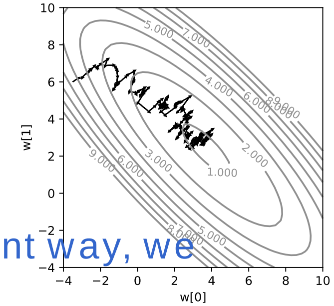
- **SGD exploits the redundancy in the samples**
  - It goes faster than full gradient in most cases
  - In practice, we use mini-batches for parallelization.







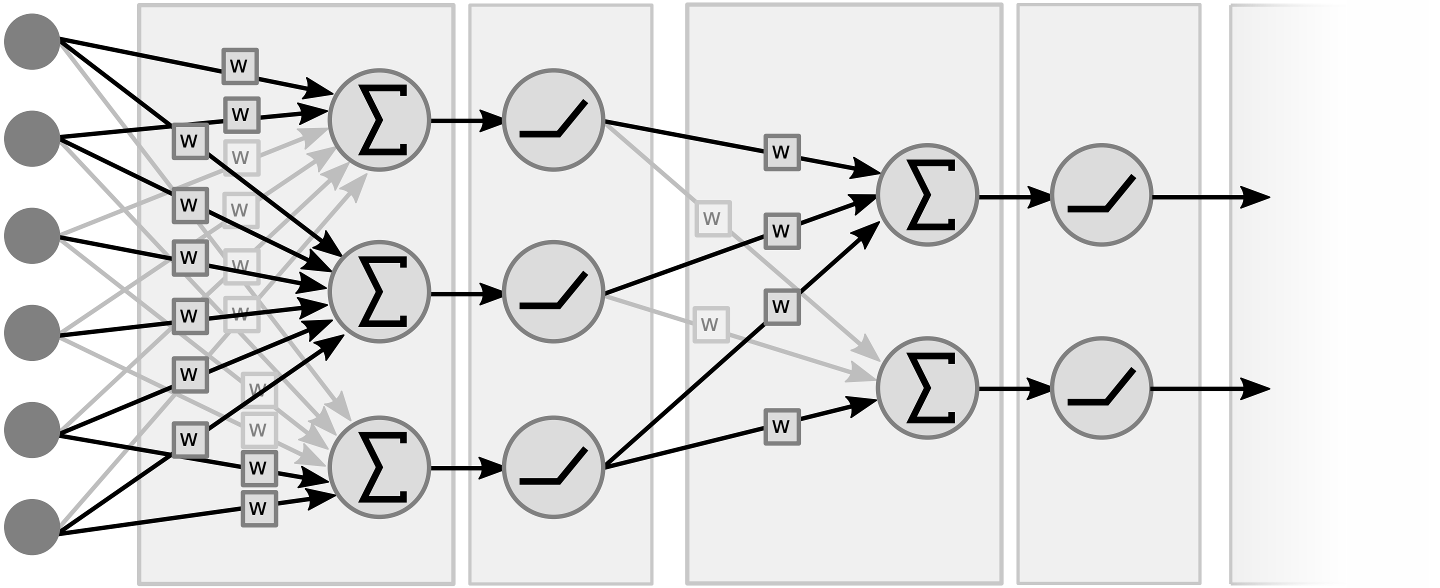




batching for parallelization is actually limited by Hardware architecture.
Ideally, if there is a new HW architecture that can do parallelization in different way, we can use that as well.

Traditional Neural Net

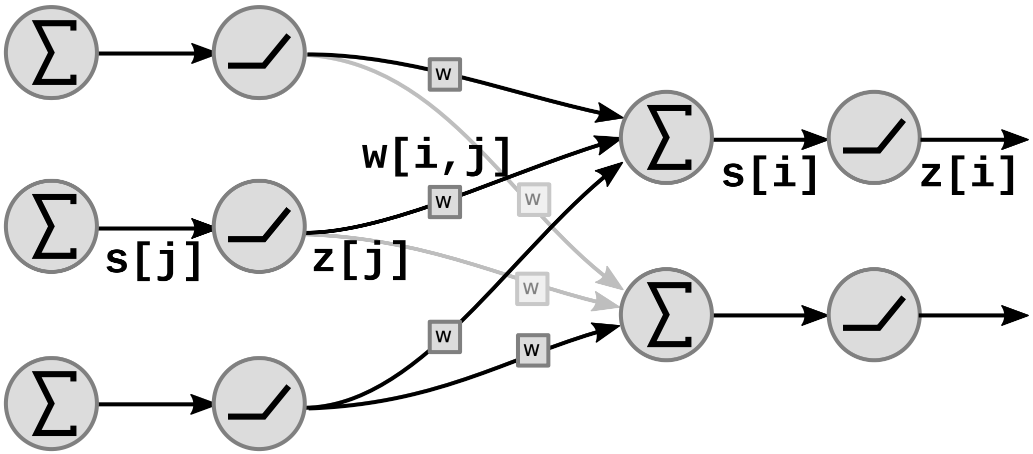
- **Stacked linear and non-linear functional blocks**
  - Weighted sums, matrix-vector product
  - Point-wise non-linearities (e.g. ReLu, tanh, ….)



Traditional Neural Net

- **Stacked linear and non-linear functional blocks**

$$s[i] = \sum_{j \in \mathrm{UP}(i)} w[i,j] \cdot z[j] \qquad z[i] = f(s[i])$$



Backprop through a non-linear function

- **Chain rule:**

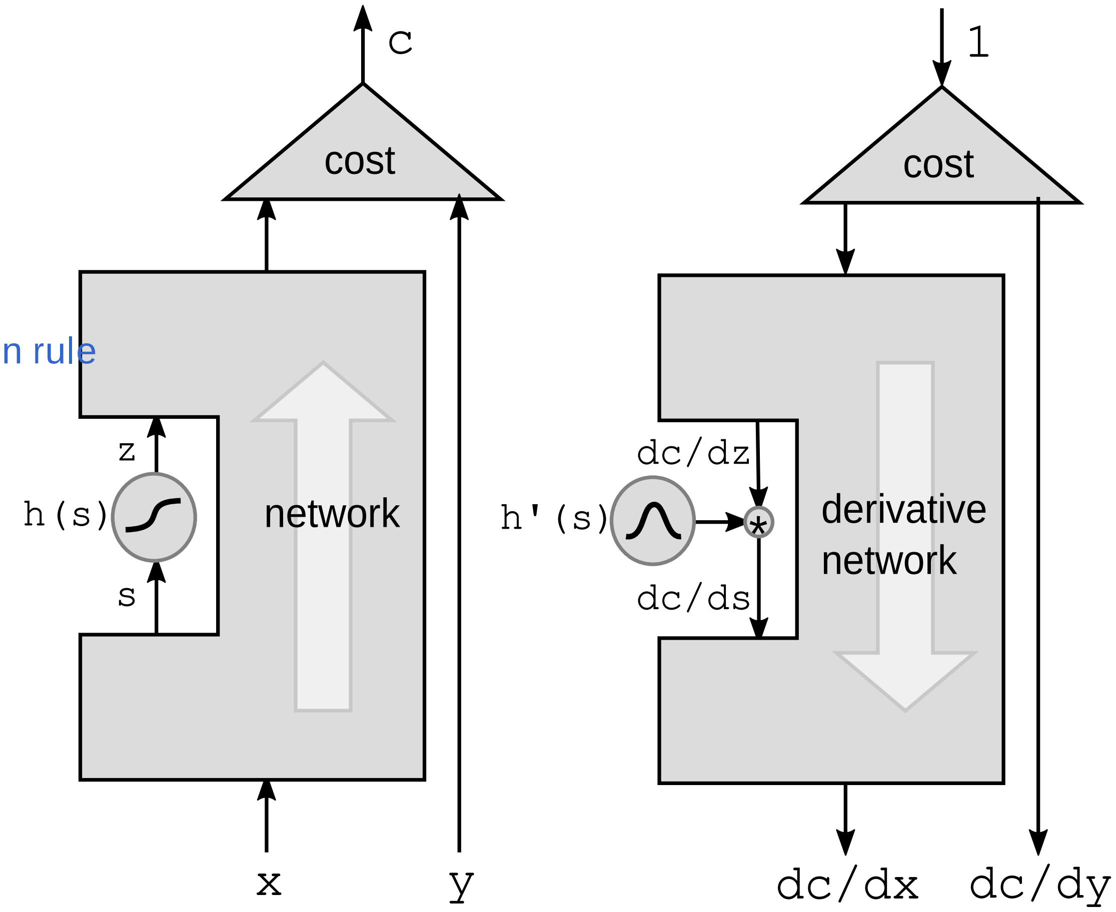
  g(h(s))’ = g’(h(s)).h’(s)

  dc/ds = dc/dz*dz/ds

  dc/ds = dc/dz*h’(s)

- **Perturbations:** Sketch to derive chain rule
  - Perturbing s by ds will perturb z by: dz=ds*h’(s)
  - This will perturb c by

    dc = dz*dc/dz = ds*h’(s)*dc/dz
  - Hence: dc/ds = dc/dz*h’(s)



Backprop through a weighted sum

- **Perturbations:**
  - Perturbing z by dz will perturb s[0],s[1],s[2] by ds[0]=w[0]*dz, ds[1]=w[1]*dz, ds[2]=w[2]*dz
  - This will perturb c by

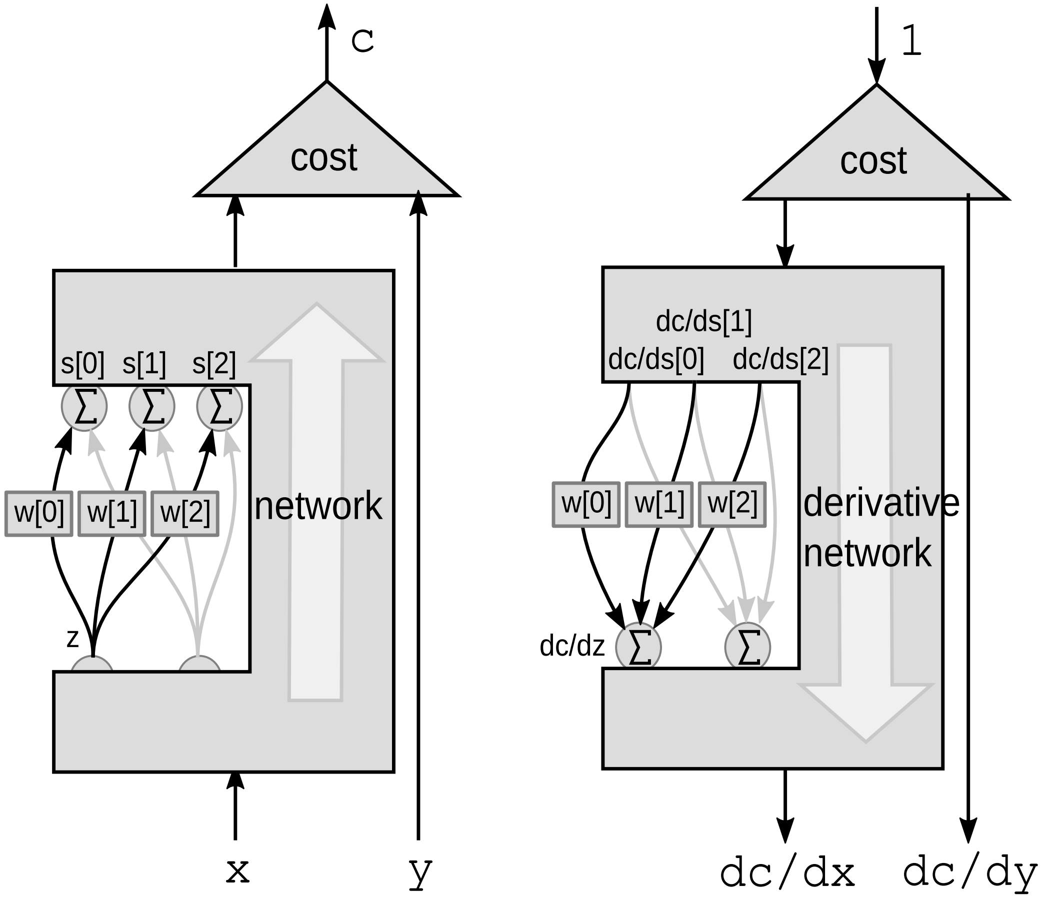
    dc = ds[0]*dc/ds[0]+

    ds[1]*dc/ds[1]+

    ds[2]*dc/ds[2]
  - Hence: dc/dz = dc/ds[0]*w[0]+

    dc/ds[1]*w[1]+

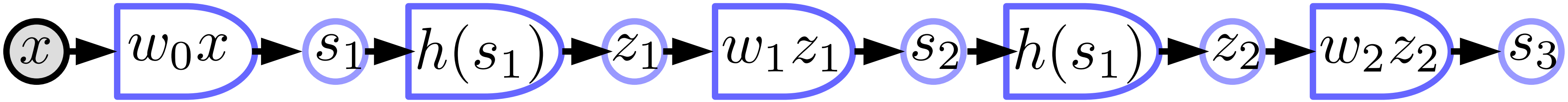
    dc/ds[2]*w[2]+



Block Diagram of a Traditional Neural Net

- **linear blocks** $s_{k+1} = w_k z_k$

- **Non-linear blocks** $z_k = h(s_k)$



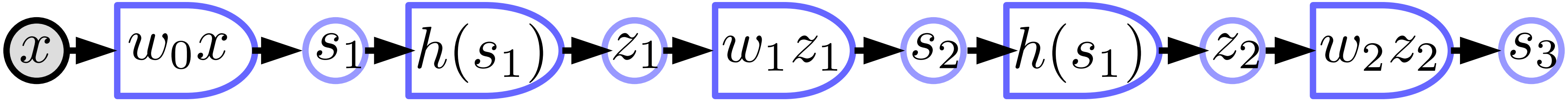
PyTorch definition

- **Object-oriented version**
  - Uses predefined nn.Linear class, (which includes a bias vector)
  - Uses torch.relu function
  - State variables are temporary

```python
import torch
from torch import nn
image = torch.randn(3, 10, 20)
d0 = image.nelement()

class mynet(nn.Module):
    def __init__(self, d0,d1,d2,d3):
        super().__init__()
        self.m0 = nn.Linear(d0, d1)
        self.m1 = nn.Linear(d1, d2)
        self.m2 = nn.Linear(d2, d3)
    def forward(self, x):
        z0 = x.view(-1)  ## flatten input tensor
        s1 = self.m0(x)
        z1 = torch.relu(s1)
        s2 = self.m1(z1)
        z2 = torch.relu(s2)
        s3 = self.m2(z2)
        return s3

model = mynet(d0,60,40,10)
out = model(image)
```



Backprop through a functional module

- **Using chain rule for vector functions**

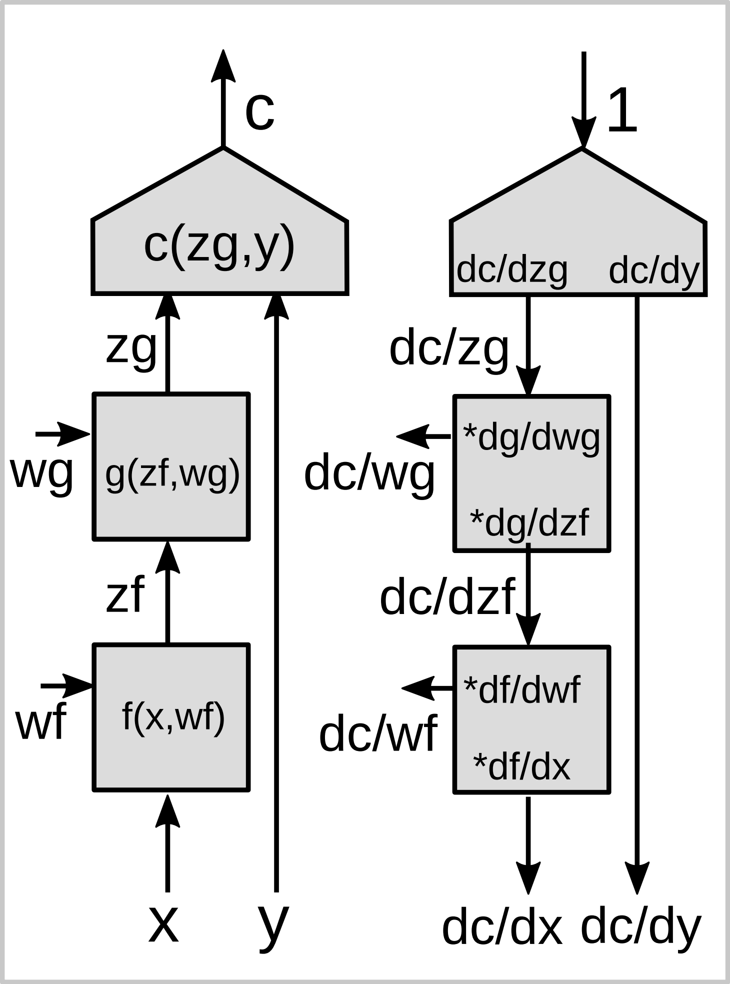
$$z_g : [d_g \times 1] \quad z_f : [d_f \times 1]$$

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\boxed{\partial z_g}} \frac{\boxed{\partial z_g}}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$

- **Jacobian matrix**
  - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f}\right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$



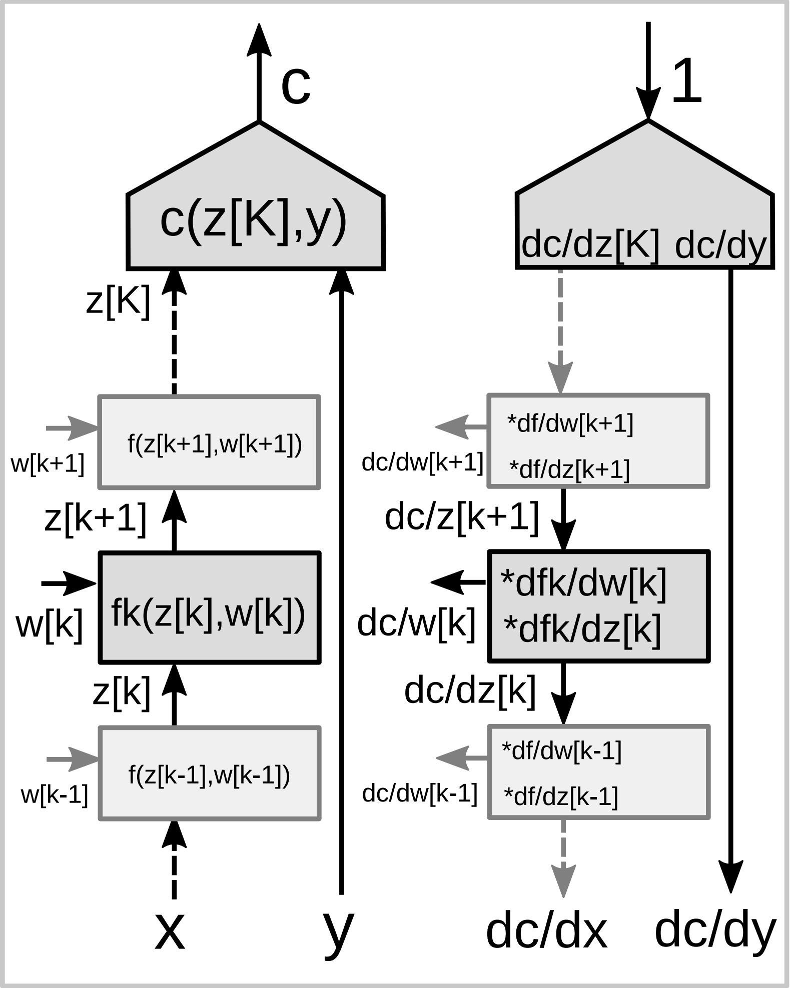
Backprop through a multi-stage graph

- **Using chain rule for vector functions**

$$\frac{\partial c}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial z_k}$$

$$\frac{\partial c}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial w_k}$$

- **Two Jacobian matrices for the module:**
  - One with respect to z[k]
  - One with respect to w[k]



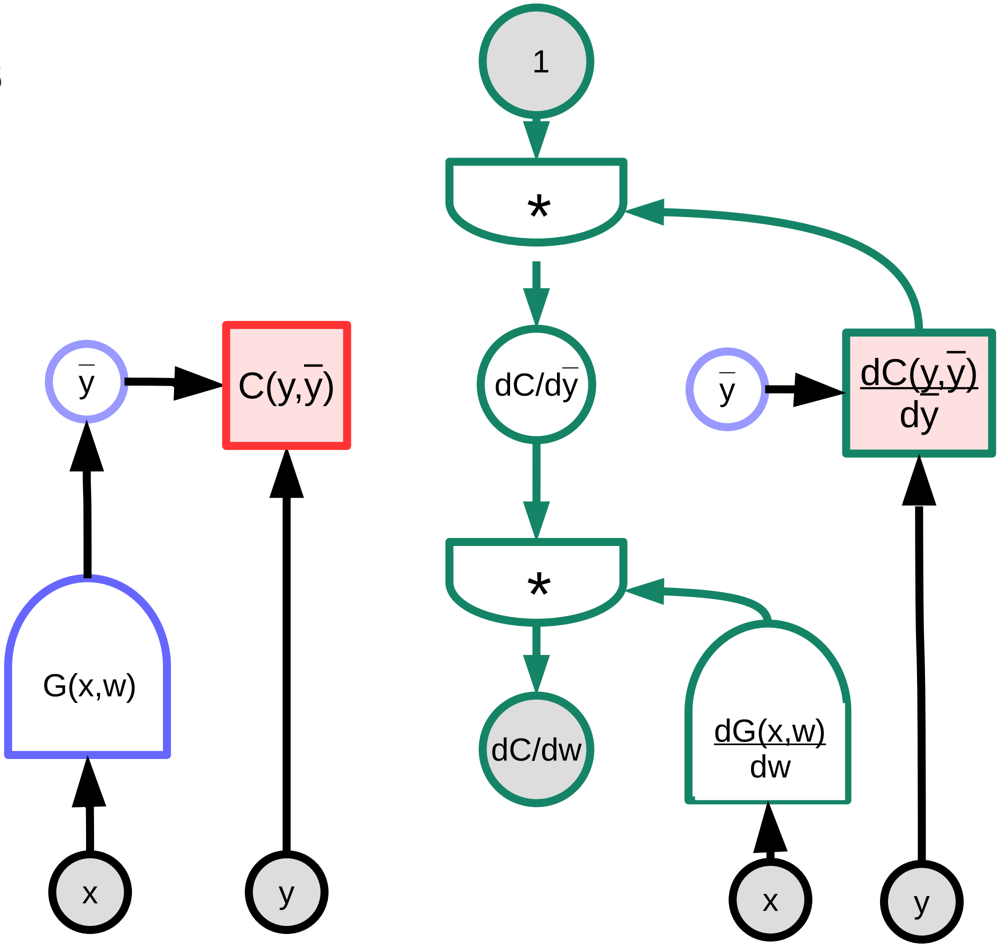
Backprop = propagation through a transformed graph

- **Derivative of composed functions**

$$C(G(w))' = C'(G(w))G'(w)$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\boxed{\partial \bar{y}}} \frac{\boxed{\partial \bar{y}}}{\partial w}$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}} \frac{\partial G(x,w)}{\partial w}$$



Y. LeCun

Gradient, Jacobian, ….

- **Dimensions:**

$$y, \bar{y} : [M \times 1] \quad w : [N \times 1]$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}} \frac{\partial \bar{y}}{\partial w}$$

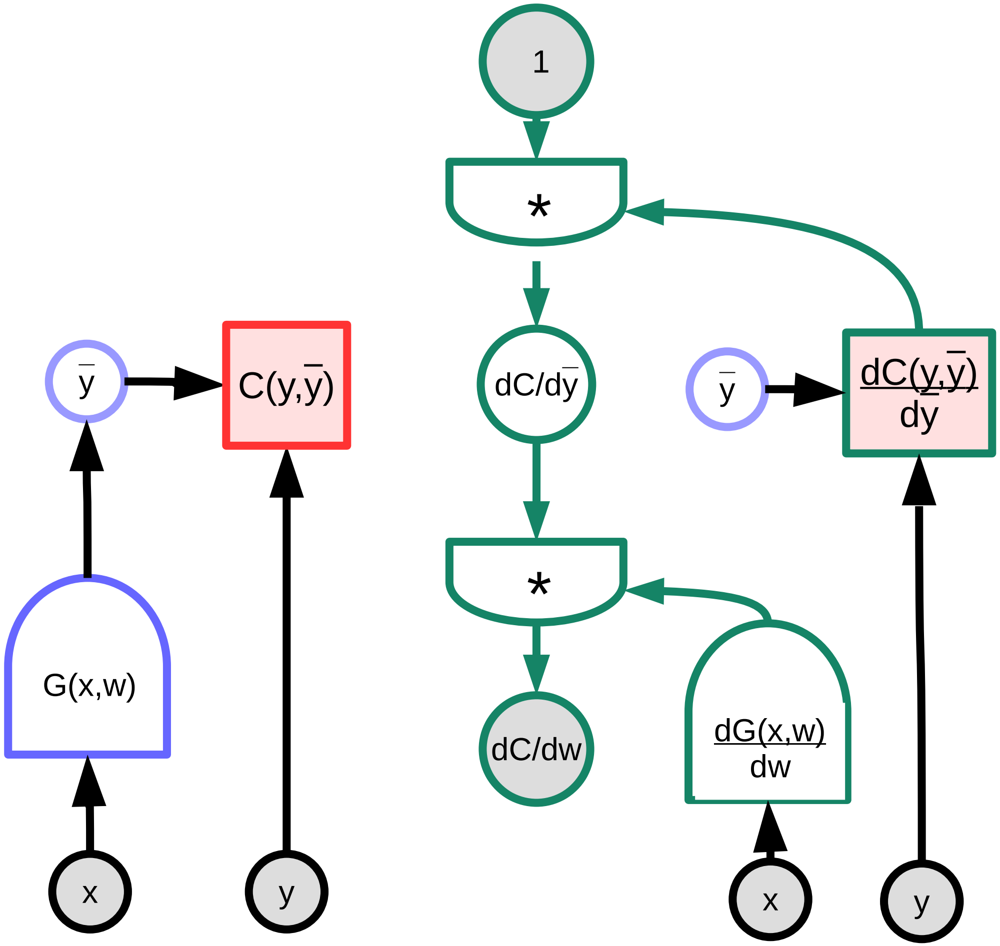
$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

- **Row vector = row vector . matrix**

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}} \frac{\partial G(x,w)}{\partial w}$$

$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

- **Gradient = gradient . Jacobian**



Basic Modules

Linear	$Y = W.X$; $dC/dX = W^T . dC/dY$; $dC/dW = dC/dY . X^T$
ReLU	$y = ReLU(x)$; if $(x<0)$ $dC/dx = 0$ else $dC/dx = dC/dy$
Duplicate	$Y1 = X, Y2 = X$; $dC/dX = dC/dY1 + dC/dY2$
Add	$Y = X1 + X2$; $dC/dX1 = dC/dY$; $dC/dX2 = dC/dY$
Max	$y = max(x1,x2)$; if $(x1>x2)$ $dC/dx1 = dC/dy$ else $dC/dx1=0$
LogSoftMax	$Yi = Xi - \log[\sum_j \exp(Xj)]$; .....???

Backprop in Practice

- **Use ReLU non-linearities (tanh and logistic are falling out of favor)**
- **Use cross-entropy loss for classification**
- **Use Stochastic Gradient Descent on minibatches**
- **Shuffle the training samples**
- **Normalize the input variables (zero mean, unit variance)**
- **Schedule to decrease the learning rate**
- **Use a bit of L1 or L2 regularization on the weights (or a combination)**
  - But it's best to turn it on after a couple of epochs
- **Use “dropout” for regularization**
  - Hinton et al 2012 http://arxiv.org/abs/1207.0580
- **Lots more in [LeCun et al. “Efficient Backprop” 1998]**
- **Lots, lots more in “Neural Networks, Tricks of the Trade” (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Müller (Springer)**

Any directed acyclic graph is OK for backprop

- **As long as there is a partial order on the modules**